co

Assignment 3

Distributed System Design

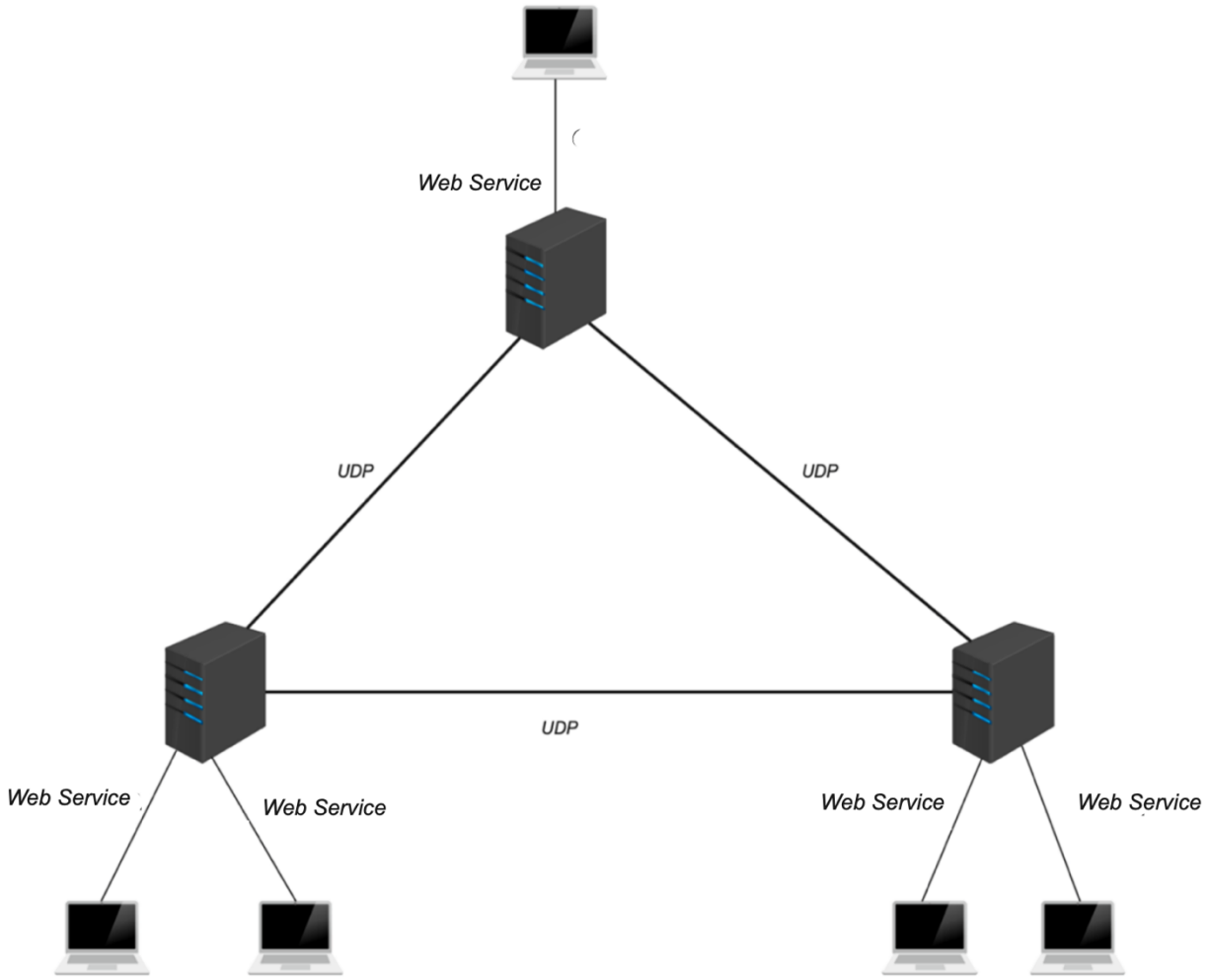
Concordia University

Peyman Shobeiri

ID: 40327586

Winter 2025

For our third assignment, we were asked to create a distributed system for share management. This system operates across three different cities, each with its server and multiple clients. Each server is managed by an administrator who can perform specific functions. The servers communicate with each other using the UDP protocol, while clients connect to their respective servers using Web Service. Additionally, each server administrator is responsible only for their own server, but the system allows clients to purchase shares from a server located in another city. An overview and general concept of the problem are shown in the figure below:



In this system, we have two types of clients: Buyers and Admins. In this system, each city has an admin, and the servers in the cities communicate with each other using UDP, and clients are connected to servers using Web Service instead of CORBA. This assignment builds on previous assignments and improves the system's reliability and consistency.

To implement this system, we first need to implement the WebInterface.java file, which defines the methods that clients (either admins or buyers) can request from the system via Web service. Next, we will implement these functions and all the necessary methods for the UDP in the ShareMarketImplementation.java file. Once that is completed, we will create a server instance that operates as a single server. Afterward, we will set up three server instances, one for each of our servers. Finally, we will develop the client file, which will include prompts and the user interface for the clients. It is important to note that our system also includes a logger file that records all necessary information for both clients and servers. Also, all of the methods in this assignment are similar to those in the second assignment.

In the image below, you can see how these operations are laid out in our updated service interface, which shows what methods can be called by users.

```java
package com.web.service;

import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)

public interface WebInterface {

        // admin
    String addShare(String shareID, String shareType, int capacity);

    String removeShare(String shareID, String shareType);

    String listShareAvailability(String shareType);



    //buyers
    String purchaseShare(String buyerID, String shareID, String shareType, int shareCount);

    String getShares(String buyerID);

    String sellShare(String buyerID, String shareID, int shareCount);

    String swapShare(String buyerID, String newShareID, String newShareType, String oldShareID, String oldShareType);

}
```

These are the function declarations without any bodies. You can find the implementation of these methods in the ShareMarketImplementation.java file. In the next step, I will explain the methods in detail for both admins and buyers.

1. Admins:

   The functions mentioned here are only visible to the admins and they are the only ones who can call these methods. So, for all of these methods, the first thing that I checked was whether the user was the admin of the specific server or not, if not, I printed an appropriate error message since only the admin of that server could perform these operations. Additionally, it is important to note that all of these methods should have the keyword synchronized since

we are using multithreading in this system. In this file, I have also created an inner class called ShareData that stores information about the shares, including the ID, type, capacity, and count of shares for each buyer ID and some basic methods, which you can see in the code.

1.1. **Add share:** In this method, I check if the share that the admin wants to add already exists in the system. If it does, we print an error message since we cannot have the same share twice otherwise, we would add it to the system. An example for calling this method is given below:

```
Admin Menu:
1. Add Share
2. Remove Share
3. List Share Availability
4. Logout
1
----------------------------------------
Please choose a shareType below:
1. Bonus
2. Equity
3. Dividend
1
----------------------------------------
Please enter the ShareID (city + M or A or E + time slot (dd+mm+yy)
tokm111122
----------------------------------------
Please enter the capacity for this new share:
6
Success: share TOKM111122 added with capacity=6
```

1.2. **Remove share:** This method is similar to the Add share method. It first checks to see if the share exists or not then if it does it attempts to remove it. One of the hard things that I have encountered here was that what if an admin wants to remove some share that exists in a buyer share list? Does it have the permission to remove that? In other words, if an admin removes a share that was purchased by a buyer, how should we handle it? I believe that since admins have more power, they could do this. So, an admin should be careful with its power since it could remove some share that belongs to the buyer.

1.3. **List share availability:** In this method, you just ask a server to write all of the available shares and some information about them. Then it will ask the other two servers to do

the same thing using the UDP protocol. So, to use it the admin chooses a share type and the selected server will print all available shares of that share type in the entire system. An example for calling this method is given below:

```
Admin Menu:
1. Add Share
2. Remove Share
3. List Share Availability
4. Logout
3
----------------------------------------
Please choose a shareType below:
1. Bonus
2. Equity
3. Dividend
1
TOKYO [Bonus]:
[ShareID=TOKM111122, Type=Bonus, Capacity=6, Purchased=5, Remaining=1]

NEWYORK [Bonus]:
[ShareID=NYKM101022, Type=Bonus, Capacity=11, Purchased=0, Remaining=11]

;LONDON [Bonus]:
No shares of type Bonus
;
```

2. Buyers' methods:

2.1. **Purchase share:** In this method, a buyer tries to buy a share from the servers. The first thing that we need to check is whether this buyer is from the same city that the server is in or not. If not, we should send this request using UDP to the target server to perform this operation there and update other servers. The hard part here was that it was a little challenging for me to work with UDP protocol and I was getting a lot of errors and that we have lots of conditions for buying from the servers like if you are buying from other cities, you cannot buy more than 3 shares per week. I have implemented all these conditions for both the other servers and the local server, allowing buyers to purchase shares from their city or others, as shown in the code. Another condition is to first check the availability of shares. If the requested share count exceeds the share's capacity, the buyer should only receive the maximum

number of shares available. Therefore, I calculate the minimum between the share remaining and the share count before attempting to buy shares. This can have four results; one is that your purchase was successful and everything is ok so you log it. Next is that that share is full and you can't get more shares and have to wait for other buyers to sell their shares so you can buy them. The next one is that the server will check and see that you have already bought this share and you cannot buy it again due to the conditions in the assignment and you get the already registered error. And the last thing is that some problems happen and you get an error like the share that you enter to buy does not exists.

```
Buyer Menu:
1. Purchase Share
2. Get Shares
3. Sell Share
4. Swap Share
5. Logout
1
----------------------------------------
Please choose a shareType below:
1. Bonus
2. Equity
3. Dividend
1
----------------------------------------
Please enter the ShareID (city + M or A or E + time slot (dd+mm+yy)
nykm101022
----------------------------------------
Please enter the number of shares:

5
Success: NYKB0002 purchased 5 of share NYKM101022
```

2.2. **Get share:** This method shows the shares that a buyer has. So, it would check the buyer ID to see if this buyer ID does exist or not, if yes then it's just 2 nested loops that print the type of this share and then the IDs for these shares and the number of shares that the buyer has. The output for this method will look like the following image:

```
Please choose an option below:
1. Purchase Share
2. Get Shares
3. Sell Share
4. Logout
2
Shares for NYKB0001:
Type [Equity]:
        NYKA121223      count: 2
Type [Bonus]:
        TOKA121222      count: 3
        NYKA111122      count: 5
```

2.3. **Sell share:** This method is called by the buyers in order to sell some shares. In the first step, I checked whether the share exists on the server or not. If not, I send a request to another server using UDP to sell that share. Next step I have checked to see if the seller does have this share or not. If this share does exist then I remove it from the seller's hash map. One challenge that I encountered was to decide whether to completely delete the share or reenter it into the system for other users to buy it. I have chosen the second one since I think that makes more sense. This way I just removed it and brought it back to the system and increase the remaining so that now other buyers could buy the sold shares. Another challenge here was that we have two maps here namely allshares and buyershare which are concurrent hash maps. So, when we sell a share in this system, we should remove them from both of these maps. Since the allshares map keeps track of all shares and its structure looks like this:

Map<String, Map<String, ShareData>> allShares;

This means that for each type of share, identified by the shareID, we have corresponding information stored as ShareData, which is an inner class that holds share information. The buyershare map shows which shares each buyer has bought with the share IDs of those shares.

2.4. **Swap share:** I have implemented this new method to allow buyers to exchange shares. When a buyer wants to swap a share the system first checks to see whether the buyer has that share or not (in the code we called it old share). It then communicates with the server hosting the new share via UDP to confirm that the new share is available in the same quantity as the old share. If both checks are successful, the system atomically completes the swap by purchasing the new share and canceling the old one. You can see an example of this method invocation in the following image.

```
Buyer Menu:
1. Purchase Share
2. Get Shares
3. Sell Share
4. Swap Share
5. Logout
4

Enter OLD share:
----------------------------------------
Please choose a shareType below:
1. Bonus
2. Equity
3. Dividend
1
----------------------------------------
Please enter the ShareID (city + M or A or E + time slot (dd+mm+yy)
nykm101022

Enter NEW share:
----------------------------------------
Please choose a shareType below:
1. Bonus
2. Equity
3. Dividend
1
----------------------------------------
Please enter the ShareID (city + M or A or E + time slot (dd+mm+yy)
tokm111122
Success: NYKB0002 swapped old share NYKM101022 with new share TOKM111122 for quantity=5
```

3. Other methods:

3.1. In this file, you can find some other methods that are methods designed for using UDP calls or simplifying basic operations. You can find more details of these methods and functions in the code.

Server Explanation (ServerInstance.java)

In the new design, we have removed the CORBA and used JAX-WS web services to implement the distributed share market system. Our ServerInstance class, located in src/server/ServerInstance.java, shows a web service endpoint instead of using an Object Request Broker (ORB). We instantiate the ShareMarketImplementation class, which contains all the logic, and then bind it to a specific URL (for example, http://localhost:8080/newyork). This makes the service accessible over SOAP to any clients that connect to that address.

Additionally, since our assignment still employs UDP for cross-server communication, we start a UDP listener thread to handle incoming requests from other city servers. Once the endpoint is

published, the server logs that it is up and running, continually listening for either SOAP requests (via JAX-WS) or UDP messages.

Client Explanation (Client.java)

The Client class, found in src/com/web/client/Client.java, remains a console-based terminal application that prompts the user for their user ID. Once the user is identified as either an admin or a buyer, the appropriate menu of operations is displayed. A significant change in this version is that we now connect using web service calls instead of CORBA stubs:

1. We create a Service object by passing in the WSDL URL and a matching QName (namespace + service name).

2. We retrieve a stub implementing our WebInterface by calling service.getPort(WebInterface.class).

3. Based on the city code in the user ID (for instance, "NYK" for New York), we select the correct endpoint stub.

Users can then choose operations such as buy, sell, swap, or list shares, and the client calls the server's web service methods (such as purchaseShare, and swapShare) over SOAP. The server responds accordingly, and the client displays the result on the console.

This approach maintains the logic of the assignment—buyers and admins can perform the same distributed operations—while leveraging JAX-WS to manage remote calls instead of CORBA.

In the last step, I created a logger that saves logs for both the servers and clients in separate files. The log files are located in the src/logs/ directory, which contains two subdirectories, one for server logs and another for client logs.

In this code, I mainly used strings for the UDP calls and separated the request string using ';' and '-'. Additionally, I mainly used concurrent hash maps, integers, and the custom classes that I have implemented and could be found in the code.

A key requirement for working with web services is that your Java and Java compiler must be version 8. Other versions may not run web services properly. To check your Java version, you can use the following commands:

java -version
javac -version

If your Java version is not 8, you can easily download or change to the appropriate version. Before running your code, then you can compile your java files and run the wsgen and wsimport commands.

I have tested this code in multiple scenarios and experienced no problems. I successfully created multiple clients, made purchases from other servers, exceeded the limits, and checked for concurrency by running multiple clients simultaneously. In all of these test cases, the code has performed well.